

An Effective Low Cost Whitebox Approach to Construct System Level Test Vectors to Detect Buffer Overflow Defects

Joe Zhou, Christopher H. Pham
ARF, Cisco Systems, Inc.
{joezhou, chpham}@cisco.com

I. Introduction

Buffer overflow continues to lead the list of 60% of the recent 2002 CERT advisories [1]. In line with the effort to solve this problem, the paper shares a low cost method as has been used effectively to construct the test vectors to detect online buffer overflow software defects. The method combines existing low cost static analysis tools with a Code-to-CLI conversion utility and a system level test configuration. The static analysis tools are used to identify potential buffer overflow at the code level. The Code-to-CLI utility provides the test engineer a set of CLI commands and options to exercise the piece of code reported by the static analysis tools. The system level test configuration facilitates the execution of the tests via the discovered CLI commands to confirm bug exhibition. The flow of the paper follows the order presented above and opens up more possibility for the reader's own applications.

II. Static Analysis (SA) and one example of buffer overflow defective code

Static Analysis is a good choice to unveil buffer overflow defects in the early development cycle[2][3]. Most SA tools will generate a high percentage warning message with high noise ratio. For huge code base exceeding a dozen million lines, the SA report can contain millions of false-positive messages. Manual efforts to review each of the messages and identify which is really a defect can be very challenging and time consuming. In most cases, extra notations should be instrumented to source code to declare boundary of array/variables to trigger the static analysis tools in order to catch the defects. This could cost more engineer hours to apply to large code base. For that reason, we implemented a lightweight "source code inspection" tool to focus only on the most buffer overflow harmful functions and ignore other types of defects.

Among the buffer overflow problems, those related to string operations usually pose higher risks. A hacker utilizes string operations such as *strcpy* could load external code to networking systems to gain system control via external system input. Therefore, it will be useful to provide the system test engineer (STE) or IT personnel a tool to catch inputs that can trigger buffer overflow. Several types of inputs are available for a typical networking device such as:

- CLI provides an interactive interface between the system and the user. There were quite a few public

reports of hacker activities thought CLI exploitation of Buffer Overflow to compromise system security [4]. One example of CLI buffer overflow exploitation is:

```
myhost:/vws/jhd/bufferoverflow/:16%a.out 1234
*****Output to Verify Initialization*****
```

```
BUF  Address      Content
buf1[0] 0x138856   a
buf1[1] 0x138857   b
buf1[2] 0x138858   c
buf2[0] 0x138859   a
buf2[1] 0x138860   b
buf2[2] 0x138861   c
*****Output to Verify Buffer Overflow*****
BUF  Address      Content
buf1[0] 0x138856   1
buf1[1] 0x138857   2
buf1[2] 0x138858   3
buf2[0] 0x138859   4           ← buffer overflow
buf2[1] 0x138860   4           ← buffer overflow
buf2[2] 0x138861   c
```

Program source code in file a.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct sample {
    char BUF1[3];
    char BUF2[3];
}sample;
sample DEST;
main (int argc, char * argv[]) {
    char *input;
    int i;
    char b[]="abc";
    for (i=0;i<3; i++){
        DEST.BUF1[i]=b[i];
        DEST.BUF2[i]=b[i];
    }
    printf ("*****Output to Verify Initialization*****\n");
    printf ("BUF\tAddress \tContent\n");
    for (i=0;i<3; i++){
        printf("buf1[%d]\t0x%d\t%c\n",i,
            &DEST.BUF1[i],DEST.BUF1[i]);
    }
    for (i=0;i<3; i++){
        printf("buf2[%d]\t0x%d\t%c\n",i,
            &DEST.BUF2[i],DEST.BUF2[i]);
    }
    strcpy(DEST.BUF1, argv[1]); /*Buffer Overflow*/
    printf ("*****Output to Verify Buffer Overflow*****\n");
    printf ("BUF\tAddress \tContent\n");
    for (i=0;i<3; i++){
        printf ("buf1[%d]\t0x%d\t%c\n",i,
            &DEST.BUF1[i],DEST.BUF1[i]);
    }
    for (i=0;i<3; i++){
        printf ("buf2[%d]\t0x%d\t%c\n",i,
            &DEST.BUF2[i],DEST.BUF2[i]);
    }
}
```

- HTTP server sometimes serves as an alternative of command input to simplify administrative tasks.
- Packets are the direct input to the networking device and may contain malformed types of inputs which trigger the vulnerable codes.

- TFTP or FTP provides a convenient way to transfer data, yet open a wide door for exploitation.

Etc.

Followings shows one of the authors' methods to help the STEs to build their input test vectors based on the CLI commands (let us use the word "test vector" to indicate any input that may potentially trigger a piece of source code containing the vulnerable functions such as one of the string operations). Due to the constraint of the paper length, we will not present our methods for other types of inputs such as packets, HTTP, TFTP, etc. Next we will show our approach for the CLI type of input.

III. Code-to-CLI conversion utility

The Code-to-CLI Conversion Utility (Code2CLI_util) designed and used by the authors has the following interfaces:

- Input: function name of the piece of code containing the vulnerable string operation.
- Output: a set of CLI commands and respective switches for that CLI. Example: in the command "*tar cvf dest source*", the CLI is *tar*, the switch is *cvf* and the inputs are *source* and *dest*.

The Code2CLI_util utilizes the CLI parser tree to derive the CLI commands and switches by reversing the CLI parser tree from the leaf source code. Notice that each OS has a CLI parser tree that grows from the root command, then fans out to all switch options down to the source code as a leaf. The authors simply wrote a utility to reverse the tree to the root level. Considering this is the classic task, explanation of the details of the reverse CLI parser tree algorithm is outside the scope of this paper and could be published in the near future to serve those who are interested.

IV. System level online detection of the buffer overflow defects

Now it is the task of the STEs or IT personnel to properly place the suggested CLI test vectors into their test scripts. Some input commands can only execute at certain privilege levels, thus extra steps should be introduced in the test scripts to get to the right state. Due to time constraints, the author could not write a utility to further automate this task, but considering it is fairly trivial, we decided to leave it to the STEs for their flexibility. However, we use the Unix grep command to scan the existing test scripts to identify the suggested CLI test vectors, thus reducing the effort to overlap more tests in future test developments.

When the test scripts are developed, the networking devices are just running as normal without any modification of existing test beds. In our environment, the test scripts look for signs of failure indicated by system crashes or illegal responses from the unit under test. Inline with our prediction, some buffer overflow defects exhibit when the proper CLI commands are issued. The result statistic is also in line with the prediction that the number of overflow defects is the same or less than that reported by the SA tool. This

confirmed the effectiveness of our combined method of using static analysis first to get a list of the vulnerable functions along with their locations in the source code and their companion CLI commands to provide the STEs a way to trigger those code segments in their test scripts. In addition, exercising the CLI's as recommended by the Code2CLI_util exempts the STEs from exhausting all CLI commands or 100% code coverage, but rather to focus on testing potential problems in a shortest achievable time.

V. Summary and Challenges

The benefit of having identified the source code where the vulnerability exists, and the companion CLI command to trigger that code segment was presented. The same method can be expanded to cover other types of inputs even though they may differ in methods of delivery. Each type of input should be treated differently thus each deserves a separate utility. Due to little dependency among the utilities (for CLI, packet, HTTP, TFTP, etc.), each can be used on-demand and not wasting the testing time if not exercised. The utilities can be run off-line thus have no effect on device performance. The STEs are now armed with more in-depth knowledge at the code level (sometimes known as white box knowledge), and still have the ability to trigger the code segment at the system level. Best of all, no expertise in source code is needed because it is quite tough sometimes to task the system level engineers to be as good as the source code developer. The reverse is also true because the source code developers most of the time do not have adequate system knowledge, yet they can still test out their code at the system level before handing it off to the device test engineers. Few challenges exist and can be the point of interest for some:

- Hybrid Operating System (OS): Although it is straight forward to develop the conversion utility for the single OS, it is not true for Hybrid OS. Thus creating the framework in the Hybrid OS environment is rather an advanced step from the base line presented in this paper.
- In large code base environments, it is more efficient to automate and provide a turn-key solution. Although it will be a challenge to provide an automatic script generator, the STEs will benefit by simply padding the generated scripts to their existing test suite.

Despite all, the baseline presented in this paper proved to save time and effort to derive a set of test vectors to counter-attack the buffer overflow vulnerabilities.

References

- [1] CERT: www.cert.org
- [2] Pham, Xu, Zhou. Source Code Analysis As A Cost Effective Practice To Improve The High Reliability of Cisco Products, Sup. Proc. IEEE ISSRE2001, pp. 144-152, 2001.
- [3] Evans. Improving Security Using Extensible Lightweight Static Analysis, <http://www.cs.virginia.edu/~evans/pubs/ieeesoftware-abstract.html>
- [4] <http://securitytracker.com/alerts/2002/Sep/1005298.html>